Completeness and Nondeterminism in Model Checking Transactional Memories *

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh

EPFL, Switzerland

Abstract. Software transactional memory (STM) offers a disciplined concurrent programming model for exploiting the parallelism of modern processor architectures. This paper presents the first *deterministic* specification automata for strict serializability and opacity in STMs. Using an antichain-based tool, we show our deterministic specifications to be equivalent to more intuitive, nondeterministic specification automata (which are too large to be determinized automatically). Using deterministic specification automata, we obtain a *complete* verification tool for STMs. We also show how to model and verify contention management within STMs. We automatically check the opacity of popular STM algorithms, such as TL2 and DSTM, with a universal contention manager. The universal contention manager is *nondeterministic* and establishes correctness for all possible contention management schemes.

1 Introduction

Software transactional memory (STM) has gained much recent interest with the advent of multicore architectures. An STM enables the programmer to structure her application in terms of coarse-grained code blocks that appear to be executed atomically [7, 12]. Behind the apparent simplicity of the STM abstraction, however, lie challenging algorithms that seek to ensure transactional atomicity without restricting parallelism. Despite the large amount of experimental work on such algorithms [8], little effort has been devoted to their formalization [3, 11].

We believe that an approach to formalizing and verifying STM algorithms can only have impact if it is accepted by the transactional memory community, and this concern has guided our decisions in choosing the correctness properties that STMs should satisfy. For this reason we consider strict serializability [9] and opacity [3] as the two measures of the correctness of STMs. The former requires committed transactions to appear as if executed at indivisible points in time during their lifetime. Opacity goes a step further and also requires aborted transactions to always access consistent state. The notion of opacity corresponds closest to an emerging consensus about correctness in the transactional software community [2, 6]. The motivation of this work is to formally check popular STM algorithms such as DSTM [6] and TL2 [2] against opacity.

* This research was supported by the Swiss National Science Foundation.

Our first step in this direction addressed the problem of space explosion in STMs [4]. We restricted our attention to STMs that satisfy certain structural properties, and we proved that the correctness of such an STM for 2 threads and 2 variables implies the correctness of the STM for an arbitrary number of threads and variables. Then, to check the correctness of an STM for 2 threads and 2 variables, we modeled an STM as a *deterministic* transition system. At the same time, we constructed *nondeterministic* specification automata for the strictly serializable and opaque words on 2 threads and 2 variables. An STM is then correct if the language of the STM transition system is included in the language of the specification automaton. Since checking language inclusion was too expensive, we resorted to checking the existence of a simulation relation. As the existence of a simulation relation is a sufficient, but not a necessary, condition for language inclusion with nondeterministic specifications, our procedure was sound but not complete.

In this paper, we provide *deterministic* specification automata for strict serializability and opacity. Constructing such deterministic specifications is nontrivial. Roughly speaking, the difficulty comes in specifying opacity in the presence of aborting transactions. In this scenario, some conflicts between transactions are transitive, whereas others are not. The determinism of the specification automata allows for an efficient check of language inclusion (by constructing the product of the specification and implementation), which results in a complete verification procedure. Moreover —and perhaps surprisingly— the deterministic specification automata are significantly smaller than their nondeterministic counterparts, which provide more intuitive specifications. As the nondeterministic automata are too large to be determinized explicitly, we use an antichain-based tool [13] to prove the correctness of our deterministic specifications. The tool shows language equivalence of our deterministic automata with the natural, nondeterministic specifications, without an explicit subset construction. The smaller, deterministic specification automata speed up the verification of STMs like DSTM and TL2 by an order of magnitude. This speed-up allows us to check the correctness of STMs with much larger state spaces. We use this gain to verify *nondeterministic* STMs that model realistic contention management schemes like exponential backoff and prioritized transactions.

In practice, STMs employ an external *contention manager* to enhance liveness [5, 10]. The idea of the contention manager is to resolve conflicts between transactions on the basis of their past behavior. Various contention managers have been proposed in the literature. For example, the *Karma* contention manager prioritizes transactions according to the number of objects opened, whereas the *Polite* contention manager backs off conflicting transactions for a random duration [10]. For verification purposes, modeling a contention manager explicitly is infeasible. First, it would blow up the state space, as the decision of a contention manager often depends on the past behavior of every thread in an intricate manner. Second, many contention managers break the structural properties that the model checking approach [4] expects in order to reduce the problem to two threads and two variables. Third, an STM is designed to maintain

safety *for all* possible contention managers, which can be changed independent of the STM.

To tackle these issues, we model the effect of all possible contention managers on an STM by defining a *universal* contention manager. An STM with the universal contention manager is a nondeterministic transition system that contains transitions for all possible decisions of any contention manager. Moreover, the universal contention manager does not break any structural property of the STM, which allows us to reduce the verification problem to two threads and two variables. Putting everything together, we are able to automatically verify opacity for STMs such as DSTM and TL2 *for all* contention managers.

Related work. This work improves the model-checking approach [4] for transactional memories in terms of both the generality of the model (including nondeterministic contention management) and the efficiency and completeness of the verification procedure. There also has been recent independent work on the formal verification of STM algorithms [1]. That verification model checks STMs applied to programs with a small number of threads and variables against the safety criteria of Scott [11], which are stronger than opacity.

2 Framework

We describe a framework to express transactions and their correctness properties.

Preliminaries. Let V be a set $\{1, \ldots, k\}$ of k *variables*, and let $C = \{\mathsf{commit}\} \cup (\{\mathsf{read}, \mathsf{write}\} \times V)$ be the set of *commands* on the variables V. Also, let $\hat{C} = C \cup \{\mathsf{abort}\}$. Let $T = \{1, \ldots, n\}$ be a set of n *threads*. Let $\hat{S} = \hat{C} \times T$ be the set of *statements*. Also, let $S = C \times T$. A word $w \in \hat{S}^*$ is a finite sequence of statements. Given a word $w \in \hat{S}^*$, we define the *thread projection* $w|_t$ of w on thread $t \in T$ as the subsequence of w consisting of all statements s in w such that $s \in \hat{C} \times \{t\}$. Given a thread projection $w|_t = s_0 \ldots s_m$ of a word w on thread t, a statement s_i is *finishing in* $w|_t$ if it is a commit or an abort. A statement s_i is *initiating in* $w|_t$ if it is the first statement in $w|_t$, or the previous statement s_{i-1} is a finishing statement.

Transactions. Given a thread projection $w|_t$ of a word w on thread t, a consecutive subsequence $x = s_0 \ldots s_m$ of $w|_t$ is a *transaction* of thread t in w if (i) s_0 is initiating in $w|_t$, and (ii) s_m is either finishing in $w|_t$, or s_m is the last statement in $w|_t$, and (iii) no other statement in x is finishing in $w|_t$. The transaction x is *committing* in w if s_m is a commit. The transaction x is *aborting* in w if s_m is an abort. Otherwise, the transaction x is *unfinished* in w. Given a word w and two transactions x and y in w (possibly of different threads), we say that x *precedes* y in w, written as $x <_w y$, if the last statement of x occurs before the first statement of y in w. A word w is *sequential* if for every pair x, y of transactions in w, either $x <_w y$ or $y <_w x$. We define a function $com : \hat{S}^* \to S^*$ such that for all words $w \in \hat{S}^*$, the word $com(w)$ is the subsequence of w which consists of every statement in w that is a part of a committing transaction. A transaction x of a thread t *writes* to a variable v if x contains a statement $((\mathsf{write}, v), t)$. A statement $s = ((\mathsf{read}, v), t)$ in x is a *global read* of a variable v if there is no statement

$((\mathsf{write}, v), t)$ before s in the transaction x. A transaction x of a thread t *globally reads* a variable v if there exists a global read of variable v in transaction x.

Correctness properties. We consider two correctness properties for transactional memories: strict serializability and opacity. Strict serializability [9] requires that the order of conflicting statements from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity, in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in STMs, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations.

A statement s_1 of transaction x and a statement s_2 of transaction y (where x is different from y) *conflict* in a word w if (i) s_1 is a global read of some variable v, and s_2 is a commit, and y writes to v, or (ii) s_1 and s_2 are both commits, and x and y write to some variable v. This notion of conflict corresponds to the deferred update semantics [8] in transactional memories, where the writes of a transaction are made global upon the commit. A word $w = s_0 \dots s_m$ is *strictly equivalent* to a word w' if (i) for every thread $t \in T$, we have $w|_t = w'|_t$, and (ii) for every pair s_i, s_j of statements in w, if s_i and s_j conflict and $i < j$, then s_i occurs before s_j in w', and (iii) for every pair x, y of transactions in w, where x is a committing or an aborting transaction, if $x <_w y$, then it is not the case that $y <_{w'} x$. We define the correctness property *strict serializability* $\pi_{ss} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to $com(w)$. Furthermore, we define *opacity* $\pi_{op} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to w. We note that $\pi_{op} \subseteq \pi_{ss}$, that is, if a word is opaque, then it is strictly serializable.

3 Transactional Memory Specifications

We capture correctness properties using TM specification automata. A *transition system* is a 3-tuple $\langle Q, q_{init}, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta \subseteq Q \times \hat{S} \times Q$ is a transition relation. A transition system is *deterministic* if for every state $q \in Q$ and every statement $s \in \hat{S}$, there is at most one state $q' \in Q$ such that $(q, s, q') \in \delta$. A word $s_0 \dots s_m$ is a *run* of the transition system if there exist states $q_0 \dots q_{m+1}$ in Q such that $q_0 = q_{init}$ and for all i such that $0 \leq i \leq m$, we have $(q_i, s_i, q_{i+1}) \in \delta$. The *language* L of a transition system is the set of all runs of the transition system. A *TM specification* Σ for a correctness property π is a transition system such that $L(\Sigma) = \pi$. A TM specification is *deterministic* if it is a deterministic transition system.

Strict serializability and opacity have been formally defined so far using nondeterministic TM specifications [4]. The nondeterminism allows a natural construction of the specification, where a transaction nondeterministically guesses a serialization point during its lifetime. A branch of the nondeterministic specification corresponds to a specific serialization choice of the transactions, which

makes the construction simple and intuitive, though redundant. Due to the nondeterminism of the specification, the existence of a simulation relation is a *sufficient but not a necessary* condition for language containment. This makes the decision procedure incomplete [4]. Moreover, these specifications are too large to be determinized automatically.

3.1 Difficulties in Providing Deterministic TM Specifications

It turns out that creating deterministic TM specifications for strict serializability and opacity is a non-trivial problem. We first give some examples that manifest the subtleties involved.

Analysis of strict serializability. We look at two words and reason whether they are strictly serializable.

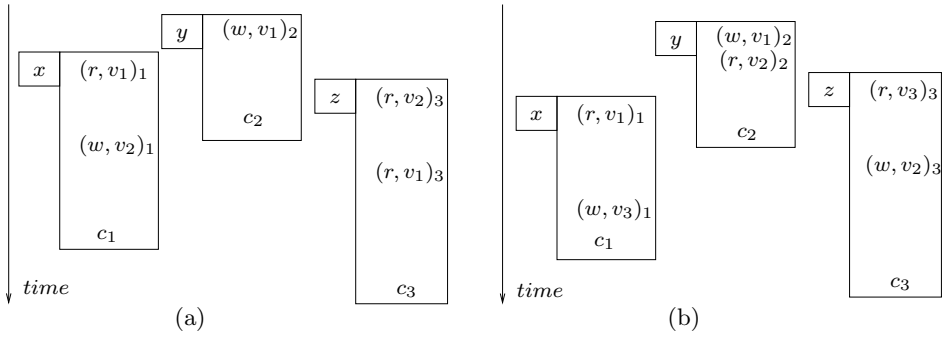

Fig. 1. Examples for strict serializability. The words are fragmented into transactions of different threads. We use the notation: w for write, r for read, c for commit, and a for abort.

- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_1), ((\mathsf{read}, v_1), t_3), (\mathsf{commit}, t_1), (\mathsf{commit}, t_3)$. The word w is illustrated in Figure 1(a). The transaction x has to serialize before y due to a conflict on v_1 (as x reads v_1 before y commits and y writes to v_1). Similarly, the transaction z has to serialize before x due to a conflict on v_2. However, z has to serialize after y due to a conflict on v_1 (z reads v_1 after v_1 is written and committed by y). So, w is not strictly serializable. On the other hand, if one of the transactions had not committed, the word would have been strictly serializable.
- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_2), t_2), ((\mathsf{read}, v_3), t_3), ((\mathsf{read}, v_1), t_1), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_3), ((\mathsf{write}, v_3), t_1), (\mathsf{commit}, t_1), (\mathsf{commit}, t_3)$. The word is illustrated in Figure 1(b). The transaction x has to serialize before y due to a conflict on v_1. Similarly, the transaction z has to serialize before x due to a conflict on v_3. Also, z writes to the variable

v_2 which is read by transaction y before z commits. Thus, z has to serialize after y. This makes w not strictly serializable.

These examples show that strict serializability is a property concerned with committing transactions. Our deterministic TM specification maintains all conflicts as part of the state. We define that a transaction x is a *weak predecessor* of transaction y in a word w if y must serialize after x for both x and y to be committing transactions. When a transaction y commits, all weak predecessors of y become weak predecessors of the threads of which y is a weak predecessor. Note that the relation weak predecessor itself is not a transitive relation. The deterministic TM specification ensures that a transaction x cannot commit if x is a weak predecessor of itself. Moreover, when a transaction commits, the information of reads and writes of the transaction has to be provided to all weak predecessors of the transaction.

Analysis of opacity. Designing a deterministic specification for opacity requires even further care. This is because even aborting transactions should be prevented from reading inconsistent values. To demonstrate the intricacies involved, we again give two examples.

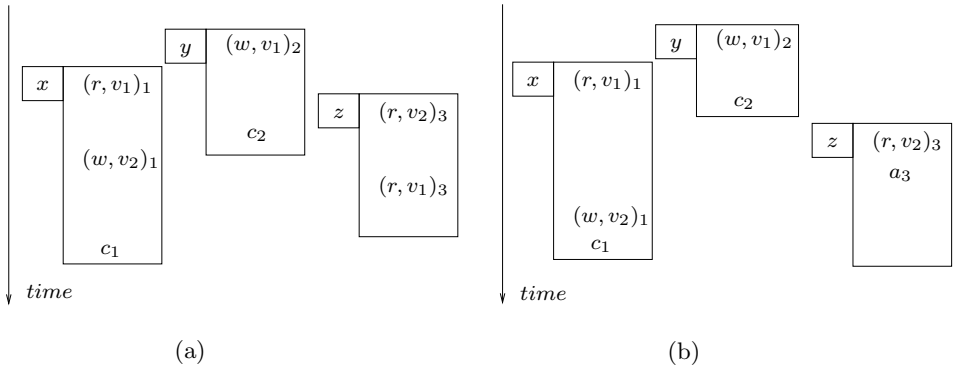

Fig. 2. Examples for opacity. The words are fragmented into transactions of different threads.

- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_2), ((\mathsf{write}, v_2), t_1), ((\mathsf{read}, v_1), t_3), (\mathsf{commit}, t_1)$. The word is illustrated in Figure 2(a). Transaction x has to serialize before y due to a conflict on v_1. Also, z has to serialize after y due to a conflict on v_1, and before x due to a conflict on v_2. Note that although z does not commit, opacity requires that transaction x does not commit. So, w is not opaque.
- Consider the word $w = ((\mathsf{write}, v_1), t_2), ((\mathsf{read}, v_1), t_1), (\mathsf{commit}, t_2), ((\mathsf{read}, v_2), t_3), (\mathsf{abort}, t_3), ((\mathsf{write}, v_2), t_1), (\mathsf{commit}, t_1)$. The word is illustrated in Figure 2(b). Transaction x has to serialize before y due to a conflict on v_1.

Transaction z has to serialize after y as they do not overlap in w. Also, z has to serialize before x due to the conflict on v_2. This makes w not opaque. This shows how a read of an aborting transaction may disallow a commit of another transaction, for the sake of opacity.

Opacity concerns committing as well as aborting transactions. Again, the deterministic TM specification for opacity maintains all conflicts as part of the state. As for strict serializability, we again use the notion of weak predecessors to store intransitive conflicts. We say that a transaction x is a *strong predecessor* of transaction y in a word w if y must serialize after x in w. Unlike weak predecessor, strong predecessor *is* a transitive relation. The specification for opacity ensures that a transaction y cannot execute *any* statement s if s makes some transaction x a strong predecessor of x. This shows how opacity poses a restriction on commands other than commit.

3.2 Deterministic TM specifications

We now present the formal definitions of the deterministic TM specifications for strict serializability and opacity. The *deterministic TM specification for strict serializability* Σ_{ss} is given by the tuple $\langle Q, q_{init}, \delta_{ss}\rangle$. A state $q \in Q$ is a 7-tuple $\langle \mathit{Status}, rs, ws, prs, pws, wp, sp\rangle$, where $\mathit{Status} : T \to \{\mathsf{started}, \mathsf{invalid}, \mathsf{pending}, \mathsf{finished}\}$ is the status, $rs : T \to 2^V$ is the read set, $ws : T \to 2^V$ is the write set, $prs : T \to 2^V$ is the prohibited read set, $pws : T \to 2^V$ is the prohibited write set, $wp : T \to 2^T$ is the weak predecessor set, and $sp : T \to 2^T$ is the strong predecessor set for the threads. If $v \in prs(t)$ (resp. $v \in pws(t)$), then the status of the thread t is set to $\mathsf{invalid}$ if t globally reads (resp. writes to) v. A thread u is in the weak predecessor set of thread t if the unfinished transaction of u is a weak predecessor of the unfinished transaction of t. The initial state q_{init} is $\langle \mathit{Status}_0, rs_0, ws_0, prs_0, pws_0, wp_0, sp_0\rangle$, where $\mathit{Status}_0(t) = \mathsf{finished}$ for all threads $t \in T$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = wp_0(t) = sp_0(t) = \emptyset$ for all threads $t \in T$. The transition relation δ_{ss} is obtained from Algorithm 1. For all states $q \in Q$ and all statements $s \in \hat{S}$, the following hold: (i) if $\mathit{specTransition}(q, s, \pi_{ss}) = \bot$, then there is no state $q' \in Q$ such that $(q, s, q') \in \delta_{ss}$, and (ii) if $\mathit{specTransition}(q, s, \pi_{ss}) = q'$ for some state $q' \in Q$, then $(q, s, q') \in \delta_{ss}$. Given a state $q = \langle \mathit{Status}, rs, ws, prs, pws, wp, sp\rangle$ and a thread $t \in T$, the procedure $\mathit{ResetState}(q, t)$ changes $\mathit{Status}(t)$ to $\mathsf{finished}$ and the sets $rs(t), ws(t), prs(t), pws(t), wp(t)$, and $sp(t)$ to $\emptyset$. The deterministic TM specification for opacity builds upon the deterministic TM specification for strict serializability. The difference comes in the strong predecessor set. We exploit the relation of strong predecessors in such a way that even aborting transactions see consistent values. For example, if a thread u is a strong predecessor of t, and t is a weak predecessor of u, then u cannot commit but t can. Many similar cases of conflict have to be carefully considered to capture the exact notion of opacity, that is, $L(\Sigma_{op}) = \pi_{op}$. The *deterministic TM specification for opacity* Σ_{op} is given by the tuple $\langle Q, q_{init}, \delta_{op}\rangle$. The set of states and the initial state are the same as those for Σ_{ss}. Also, the transition relation δ_{op} can be similarly obtained from Algorithm 1 using the property π_{op} in place of π_{ss}.

Algorithm 1 *specTransition*$(\langle Status, rs, ws, prs, pws, wp, sp\rangle, s, \pi)$

if $s = ((\mathsf{read}, v), t)$ **then**
  if $v \in ws(t)$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
  if $\pi = \pi_{op}$ **then**
    $U := \{u \in T \mid v \in prs(u)$ or $v \in prs(u')$ such that $u \in sp(u')\}$
    if $t \in U$ or there exists a thread $u \in U$ such that $t \in sp(u)$ **then return** $\bot$
  if $Status(t) = \mathsf{finished}$ **then**
    add all threads $u \in T$ such that $Status(u) = \mathsf{pending}$ to $wp(t)$ and $sp(t)$
    add all threads $u' \in T$ to $sp(t)$ such that $u' \in sp(u)$ and $Status(u) = \mathsf{pending}$
    $Status(t) := \mathsf{started}$
  $rs(t) := rs(t) \cup \{v\}$
  if $v \in prs(t)$ **then** $Status(t) := \mathsf{aborted}$
  for all threads $u \in T$ **do**
    if $v \in ws(u)$ **then** $wp(u) := wp(u) \cup \{t\}$
    if $v \in prs(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
  if $\pi = \pi_{ss}$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
  for all threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do** $sp(u) := sp(u) \cup U$
  for all threads $u \in T$ such that $u \in sp(t)$ **do**
    $pws(u) := pws(u) \cup \{v\}$
    if $v \in ws(u)$ **then** $Status(u) := \mathsf{aborted}$
if $s = ((\mathsf{write}, v), t)$ **then**
  if $Status(t) = \mathsf{finished}$ **then**
    add all threads $u \in T$ such that $Status(u) = \mathsf{pending}$ to $wp(t)$ and $sp(t)$
    add all threads $u' \in T$ to $sp(t)$ such that $u' \in sp(u)$ and $Status(u) = \mathsf{pending}$
    $Status(t) := \mathsf{started}$
  $ws(t) := ws(t) \cup \{v\}$
  if $v \in pws(t)$ **then** $Status(t) := \mathsf{aborted}$
  for all threads $u \in T$ **do**
    if $v \in rs(u)$ **then**
      $wp(t) := wp(t) \cup \{u\}$
      if $\pi = \pi_{op}$ **and** $t \in sp(u)$ **then** $Status(t) := \mathsf{aborted}$
    if $v \in pws(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
if $s = (\mathsf{commit}, t)$ **then**
  if $t \in wp(t)$ **then return** $\bot$
  if $\pi = \pi_{op}$ **then**
    $U := \{u \mid u \in wp(t)$ or $u \in sp(u')$ for some $u' \in wp(t)\}$
    if $t \in U$ or there exists a thread $u \in U$ such that $t \in sp(u)$ **then return** $\bot$
  for all threads $u \in T$ such that $u \in wp(t)$ **do**
    if $ws(u) \cap ws(t) \neq \emptyset$ **then** $Status(u) := \mathsf{aborted}$ **else** $Status(u) := \mathsf{pending}$
    $prs(u) := prs(u) \cup prs(t) \cup ws(t)$
    $pws(u) := pws(u) \cup pws(t) \cup ws(t) \cup rs(t)$
    for all threads $u' \in T$ such that $t \in wp(u')$ or $ws(u') \cap ws(t) \neq \emptyset$ **do**
      $wp(u') := wp(u') \cup \{u\}$
  for all threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do** $sp(u) := sp(u) \cup U$
  ResetState(q, t)
if $s = (\mathsf{abort}, t)$ **then** *ResetState*(q, t)
return $\langle Status, rs, ws, prs, pws, wp, sp\rangle$

3.3 Model Checking with Deterministic TM Specifications

It has been shown [4] that for a transactional memory which satisfies certain structural properties, it is sufficient to show its correctness for all programs with two threads and two variables in order to prove the correctness of the transactional memory for all programs. These properties were shown for transactional memories like DSTM [6] and TL2 [2]. The nondeterministic TM specifications presented [4] are too huge to be automatically determinized. However, surprisingly enough, the deterministic TM specifications we present in this paper turn out to be much smaller in size. Using an antichain-based tool [13], we establish that for two threads and two variables, the language of our deterministic TM specification for strict serializability (resp. opacity) is equivalent to the language of the nondeterministic specification for strict serializability (resp. opacity) [4].

For strict serializability, our deterministic TM specification Σ_{ss} has only 3520 states, whereas the nondeterministic one A_{ss} has 12345 states. Similarly, for opacity, Σ_{op} has 2272 states, while the nondeterministic specification A_{op} requires 9202 states. Moreover, the deterministic TM specifications allow for an efficient procedure that directly checks, whether the language of the TM algorithm is included in the language of the deterministic TM specifications. This procedure makes our model checking complete too. We show the results in Table 1. For deterministic STMs [4], we observe that checking language inclusion with deterministic TM specifications is much faster than checking existence of a simulation relation with nondeterministic TM specifications.

4 Nondeterministic Transactional Memories

Our succinct deterministic TM specifications tempt us to go a step further in model checking transactional memories. Transactional memories often employ nondeterministic schemes to resolve conflicts, in the face of thread failures or repetitive aborts of a thread. These schemes are generally treated externally to the transactional memory, and are referred to as contention managers. The notion of a contention manager helps to keep the design of a transactional memory modular. This allows a transactional memory to switch from one contention manager to another, depending upon the contention scenario [5]. An STM is designed in such a way that it maintains its correctness property for all possible contention managers.

Transactional memories have been modeled in a restrictive framework as TM algorithms [4], where a transactional memory is tied to an implicit, *specific* contention manager. We now give a general formalism which is practically more useful, where a transactional memory is separated from the contention manager.

4.1 A Formalism for TM with Contention Managers

Programs. We express a thread program as an infinite binary trees on commands. For every command of a thread, we define two successor commands, one

Table 1. Time for simulation (resp. language inclusion) checking for STMs on a quad dual core 2.8 GHz server with 16 GB RAM. In case simulation (resp. language inclusion) holds, we write Y followed by the time required for finding it. Otherwise, we write N followed by the counterexample produced, followed by the time required to prove that no simulation exists (resp. language inclusion does not hold), followed by the time required to find the counterexample. A '*' for the search for simulation relation means that it does not complete in 2 hours, but we do find a counterexample. A '–' means that the search for both, the simulation relation and the counterexample, does not complete in 2 hours.

TM algorithm A	Number of states	$A \prec A_{ss}$	$A \prec A_{op}$	$L(A) \subseteq L(\Sigma_{ss})$	$L(A) \subseteq L(\Sigma_{op})$
Deterministic STMs [4]					
seq	3	Y, $0.8s$	Y, $0.7s$	Y, $0.01s$	Y, $0.01s$
2PL	99	Y, $13s$	Y, $8s$	Y, $0.01s$	Y, $0.01s$
dstm	944	Y, $127s$	Y, $82s$	Y, $0.09s$	Y, $0.07s$
TL2	11840	Y, $1647s$	Y, $1438s$	Y, $1.2s$	Y, $1s$
occ	4480	Y, $765s$	N, w_1, $567s$,$4s$	Y, $0.46s$	N, w_1, $0.41s$, $4s$
TL2 mod.	17520	N, w_2, *, $9s$	N, w_2, *, $9s$	N, w_2, $2.7s$, $9s$	N, w_2, $2.1s$, $8s$
Nondeterministic STMs					
dstm	1846	Y, $303s$	Y, $279s$	Y, $0.16s$	Y, $0.13s$
TL2	21568	–	–	Y, $3.2s$	Y, $2.4s$
Counterexamples					
w_1	$(w,1)_2, (r,1)_1, c_2, (r,1)_1$				
w_2	$(w,2)_1, (w,1)_2, (r,2)_2, (r,1)_1, c_2, c_1$				

if the command is successfully executed, and another if the command fails due to an abort of the transaction. We use a set of thread programs to define a multithreaded program. Formally, a *thread program* θ on a set C of commands is a function $\theta : \mathbb{B}^* \to C$. We define a *(multithreaded) program* p on n threads and k variables as an n-tuple $p = \langle \theta^1, \ldots, \theta^n \rangle$ of thread programs on C.

TM algorithms. We model transactional memories using TM algorithms. A TM algorithm consists of a set of states, an initial state, an extended set of commands depending on the underlying TM, a conflict function, a pending function, and a transition relation between the states. The extended commands include the set C of commands, and TM specific additional commands. For example, a given TM may require that a thread locks a variable before writing to the variable. Every extended command is assumed to execute atomically. The conflict function captures the statements in the states, when the TM algorithm needs to consult a contention manager for a decision. The pending function represents the pending command of a thread in a state, and ensures that if a thread has not finished the execution of a particular command, then no other command is executed by the thread.

We define a *TM algorithm* $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, D is the set of extended commands with $C \subseteq D$, $\phi : Q \times D \to \mathbb{B}$ is the conflict function, $\gamma : Q \times T \to C \cup \{\bot\}$ is the pending function, and $\delta \subseteq Q \times \hat{C} \times \hat{S}_D \times \mathit{Resp} \times Q$ is the transition relation, where $\hat{S}_D = (D \cup \{\mathsf{abort}\}) \times T$ and $\mathit{Resp} = \{\bot, 0, 1\}$. For a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, the following rules hold:

- For all threads $t \in T$, we have $\gamma(q_{init}, t) = \bot$.
- For all states $q, q' \in Q$ such that there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, if $r = \bot$, then $\gamma(q', t) = c$, otherwise $\gamma(q', t) = \bot$.
- For all states $q, q' \in Q$ such that there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, then $\gamma(q', u) = \gamma(q, u)$ for all threads $u \neq t$.
- For all states q and all threads t, if $\gamma(q, t) = c$ where $c \neq \bot$, then for all outgoing transitions $(q, c_1, (d, t), r, q')$ from q in δ, we have $c_1 = c$.
- For all states q and all threads t, if $\gamma(q, t) = \bot$, then there is an outgoing transition $(q, c, (d, t), r, q'$ from q in δ for every command $c \in C$.
- For all $q \in Q$, for all transitions $(q, c, (d, t), r, q')$ in δ, we have $d = \mathsf{abort}$ if and only if $r = 0$.

Note that the rules above restrict the transition relation δ and the pending function γ such that γ is unique. A command c is *enabled* in a state q for thread t if $\gamma(q, t) \in \{\bot, c\}$ (i.e., either no command is pending, or c itself is pending). A command c is *abort enabled* in a state q for thread t if c is enabled in q for thread t and there is no transition $(q, c, (d, t), r, q') \in \delta$ such that $d \in D$. A transition relation δ is *deterministic* if for all $q \in Q$ and $(c, t) \in S$, if $(q, c, (d_1, t), r_1, q_1) \in \delta$ and $(q, c, (d_2, t), r_2, q_2) \in \delta$, then $d_1 = d_2$, $r_1 = r_2$, and $q_1 = q_2$. A TM algorithm is *deterministic* if its transition relation is deterministic.

Contention managers. When the transactional memory detects a conflict (the conflict function is true), it requests the contention manager to resolve the conflict. The contention manager proposes the TM algorithm the next statement to be executed. Formally, a *contention manager cm* on a set D of commands is a function $cm : \hat{S}_D^* \to 2^{\hat{S}_D}$, such that if the last statement of w is from thread t, then every statement in $cm(w)$ is a statement of t.

Given a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$ and a contention manager $cm : \hat{S}_D^* \to 2^{\hat{S}_D}$, we define a *TM algorithm and contention manager pair* $\langle M, cm \rangle = \langle Q_\times, (q_{init}, \varepsilon), D, \gamma_\times, \delta_\times \rangle$, where $Q_\times = Q \times \hat{S}_D^*$ is the set of states, $\gamma_\times : Q_\times \times T \to C \cup \{\bot\}$ is the pending function such that for all states $q_\times \in Q_\times$ and all threads $t \in T$, we have $\gamma_\times(q_\times, t) = \gamma(q, t)$ where $q_\times = (q, w)$ for some word $w \in \hat{S}_D^*$, $\delta_\times \subseteq Q_\times \times \hat{C} \times \hat{S}_D \times \mathit{Resp} \times Q_\times$ is the transition relation such that for all states $q_\times, q'_\times \in Q_\times$, for all commands $c \in \hat{C}$, for all statements $s \in \hat{S}_D$, and for all responses $r \in \mathit{Resp}$, we have $(q_\times, c, s, r, q'_\times) \in \delta_\times$ if and only if (i) there is a transition $(q, c, s, r, q') \in \delta$, and (ii) if $\phi(q, s) = \mathsf{true}$, then $s \in cm(w)$, where $w \in \hat{S}_D^*$ and $q, q' \in Q$ such that $q_\times = (q, w)$ and $q'_\times = (q', w \cdot s)$.

Runs and languages of TM algorithms. On putting the pieces together, a TM algorithm interacts with a program, a scheduler, and a contention manager (see Fig. 3). A thread of the program is chosen by the scheduler, and the next

command of the thread is given to the TM algorithm. The TM algorithm decides whether the command can be executed in a single or several atomic steps, or the command is in conflict. The commands executed by the TM algorithm are also reported to the contention manager for its bookkeeping. If the TM algorithm finds a conflict, the TM algorithm resolves the conflict using the contention manager. The TM algorithm makes a transition accordingly, and gives back to the program a response. The response is $\bot$ if the TM algorithm needs additional steps to complete the command, 0 if the TM algorithm needs to abort the transaction of the scheduled thread, and 1 if the TM algorithm has completed the command. Given a program, a scheduler, a TM algorithm, and a contention manager, we get a run. Projecting the run to the set of successful statements (that is, aborts, and statements that get response 1) gives an infinite word. The language of a TM algorithm and contention manager pair is the set of infinite words that the TM algorithm can produce for any program and any scheduler, where conflicts are resolved using the specific contention manager.

Formally, a *scheduler* σ on T is a function $\sigma : \mathbb{N} \to T$. Let $p = \langle \theta^1, \ldots, \theta^n \rangle$ be a program, and let σ be a scheduler. A *run* $\rho = \langle q_0, l_0, (d_0, t_0), r_0 \rangle \langle q_1, l_1, (d_1, t_1), r_1 \rangle \ldots$ of a TM algorithm A with scheduler σ on program p and contention manager cm is an infinite sequence of tuples of states, program locations, statements, and responses, where $l_j = \langle l_j^1, \ldots, l_j^n \rangle \in (\mathbb{B}^*)^n$ for all $j \geq 0$ and the following hold: (i) $q_0 = q_{init}$ and $l_0 = \langle \varepsilon, \ldots, \varepsilon \rangle$, and (ii) for all $j \geq 0$, there exists a transition $(q_j, c_j, (d_j, t_j), r_j, q_{j+1}) \in \delta$ such that if $\phi(q_j, (d_j, t_j)) = \mathsf{true}$, then $(d_j, t_j) \in cm((d_0, t_0) \ldots (d_{j-1}, t_{j-1}])$, and (iii) $t_j = \sigma(j)$, and (iv) $c_j = \theta^{t_j}(l_j^{t_j})$, and (v) for all $t \in T$, we have $l_{j+1}^t = l_j^t$ if either $t \neq t_j$ or $r_j = \bot$, and $l_{j+1}^t = l_j^t \cdot r_j$ otherwise. A statement $s_j \in \hat{S}$ is *successful* in the run $\rho = \langle q_0, l_0, s_0, r_0 \rangle \langle q_1, l_1, s_1, r_1 \rangle \ldots$ if (i) $r_j \in \{0, 1\}$, or (ii) $r_k = 1$ with $j < k$ and $r_{j+1} \ldots r_{k-1}$ are all equal to $\bot$. We define the *language* $L(\langle A, cm \rangle)$ of a $\langle A, cm \rangle$ pair as the set of all infinite words $w \in \hat{S}^\omega$ such that w is the sequence of all successful statements in a run of A with some scheduler on some program and the contention manager cm. A TM algorithm A with a contention manager cm ensures a correctness property $\pi \subseteq \hat{S}^*$ if every finite prefix of every word in $L(\langle A, cm \rangle)$ is in π.

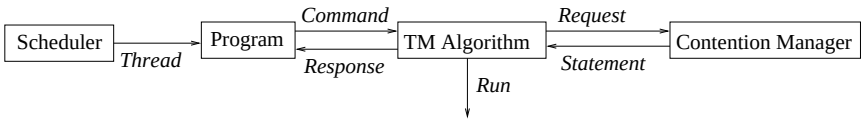

Fig. 3. Interaction in the model

Modeling contention managers explicitly in our formalism is not a feasible option. First of all, contention managers may blow up the state space as their decisions may depend intricately on past behavior. For example, a simple random backoff contention manager, that asks a conflicting thread to back off for a random amount of time could blow up the state space. Secondly, some of the

structural properties break when we model a TM algorithm in conjunction with a particular contention manager. For example, if a contention manager prioritizes transactions according to the number of times it has aborted in the past, then the TM algorithm does not satisfy the structural property of 'transactional projection' [4]. This is because, an abort of a transaction of thread t may be the reason why the next transaction of thread t commits. As the remaining structural properties build upon the transactional projection property, they also collapse for specific contention managers.

We take a novel approach to model check transactional memories with different contention managers. Given a TM algorithm A with extended alphabet D, we define a *universal contention manager ucm* such that for all words $w \in \hat{S}_D^*$, we have $ucm(w) = \hat{S}_D$. The idea of the universal contention manager is to allow nondeterministically all choices that the TM algorithm has. It is easy to observe that the transition relation for the pair $\langle A, ucm\rangle$ is identical to that of the TM algorithm A. From the definition of the language of a TM algorithm and a contention manager pair, we get $L(\langle A, cm\rangle) \subseteq L(\langle A, ucm\rangle)$ for every contention manager cm. Thus, if a TM algorithm ensures a correctness property with the universal contention manager, then the TM algorithm is correct *for all* contention managers. Moreover, if a TM algorithm A satisfies the structural properties, then the pair $\langle A, ucm\rangle$ also satisfies the structural properties [4]. Thus, verifying the correctness of the TM algorithm with ucm for two threads and two variables proves the correctness of the TM algorithm for arbitrary number of threads and variables for all possible contention managers.

We now provide, as examples, nondeterministic DSTM and nondeterministic TL2, combined with the universal contention manager. We then verify their correctness.

4.2 Nondeterministic DSTM

Dynamic software transactional memory (DSTM) [6] is one of the most popular transactional memories. DSTM faces a conflict when a transaction wants to own a variable which is owned by another thread. We define the nondeterministic DSTM algorithm A_{dstm} as $\langle Q, q_{init}, D, \gamma, \delta_{dstm}\rangle$. A state $q \in Q$ is defined as a 3-tuple $\langle Status, rs, os\rangle$, where $Status : T \to \{\mathsf{aborted}, \mathsf{validated}, \mathsf{invalid}, \mathsf{finished}\}$ is the status function, and $rs : T \to V$ is the read set, and $os : T \to V$ is the ownership set.

The initial state $q_{init} = \langle Status_0, rs_0, os_0\rangle$, where for all threads $t \in T$, we have $Status_0(t) = \mathsf{finished}$ and $rs_0(t) = os_0(t) = \emptyset$. The set of extended commands is $D = C \cup (\{\mathsf{own}\} \times V) \cup \{\mathsf{validate}\}$. The transition relation δ_{dstm} is obtained from Algorithm 2. For all states $q \in Q$, all commands $c \in C$, all extended commands $d \in D \cup \{\mathsf{abort}\}$, all threads $t \in T$, and all responses $r \in Resp$, we have: (i) if $dstmTransition(q, c, d, t, r) = \bot$, then there does not exist a state $q' \in Q$ such that $(q, c, (d, t), r, q') \in \delta_{dstm}$, and (ii) if $dstmTransition(q, c, d, t, r) = q'$ for some state $q' \in Q$, then $(q, c, (d, t), r, q') \in \delta_{dstm}$.

Our second example is a model of another popular transactional memory, transactional locking 2 (TL2) [2] with the universal contention manager. We give

Algorithm 2 $dstmTransition(\langle Status, rs, os\rangle, c, d, t, r)$

if c is not enabled in q for thread t **then return** $\bot$
if $c = (\mathsf{read}, v)$ **then**
  if $d = c$ **and** $v \in os(t)$ **and** $r = 1$ **and** $Status(t) \neq \mathsf{aborted}$ **then return** q
  if $d = c$ **and** $v \notin os(t)$ **and** $r = 1$ **and** $Status(t) = \mathsf{finished}$ **then**
    $rs(t) := rs(t) \cup \{v\}$
    return q
if $c = (\mathsf{write}, v)$ **then**
  if $d = c$ **and** $v \in os(t)$ **and** $r = 1$ **and** $Status(t) \neq \mathsf{aborted}$ **then return** q
  if $d = (\mathsf{own}, v)$ **and** $r = \bot$ **and** $Status(t) \neq \mathsf{aborted}$ **then**
    $os(t) := os(t) \cup \{v\}$
    for all threads $u \neq t$ such that $v \in os(u)$ **do**
      $Status(u) := \mathsf{aborted} \quad rs(u) := \emptyset \quad os(u) := \emptyset$
    return q
if $c = \mathsf{commit}$ **then**
  if $d = \mathsf{validate}$ **and** $r = \bot$ **and** $Status(t) = \mathsf{finished}$ **then**
    $Status(t) := \mathsf{validated}$
    for all threads $u \neq t$ such that $rs(t) \cap os(u) \neq \emptyset$ **do**
      $Status(u) := \mathsf{aborted} \quad rs(u) := \emptyset \quad os(u) := \emptyset$
    return q
  if $d = c$ **and** $r = 1$ **and** $Status(t) = \mathsf{validated}$ **then**
    $Status(t) := \mathsf{finished} \quad rs(t) := \emptyset \quad os(t) := \emptyset$
    for all threads $u \neq t$ such that $rs(u) \cap os(t) \neq \emptyset$ **do** $Status(u) := \mathsf{invalid}$
    return q
if $d = \mathsf{abort}$ **and** $r = 0$ **then**
  $Status(t) := \mathsf{finished} \quad rs(t) := \emptyset \quad os(t) := \emptyset$
  if c is abort enabled in q **and** $d = \mathsf{abort}$ **and** $r = 0$ **then return** q
  if $c = (\mathsf{write}, v)$ **and** $v \notin os(t)$ **and** $v \in os(u)$ s.t. $u \neq t$ **then return** q
  if $c = \mathsf{commit}$ **and** $Status(t) = \mathsf{finished}$ **and** $rs(t) \cap os(u) \neq \emptyset$ s.t. $u \neq t$ **then**
    return q
return $\bot$

an informal description of the role of *ucm* in TL2. TL2 uses locks for ensuring opacity. A thread locks all the variables in the write set at the time of commit. With TL2 algorithm using the universal contention manager, whenever a thread t conflicts due to a variable being locked by another thread u, the nondeterministic TL2 algorithm has the following transitions: one to abort t, and others to allow the thread t to proceed by setting the abort flag of some thread u.

We note that nondeterministic DSTM and nondeterministic TL2, combined with the universal contention manager satisfy the transactional projection property, as aborting or unfinished transactions can influence committing transactions only by forcing them to abort. The remaining structural properties depend on the transactional projection property, but are not influenced by a contention manager. Thus, all required structural properties do hold for nondeterministic DSTM and nondeterministic TL2 obtained with the universal contention manager. We check whether the language of these nondeterministic STMs is included in the language of the deterministic TM specifications. Our results, shown in Ta-

ble 1, establish that DSTM and TL2 ensure opacity for an arbitrary number of threads and variables for all contention managers. We observe that the number of states in the nondeterministic TM algorithm using the universal contention manager is nearly double the number of states in the corresponding deterministic TM algorithm. We note that the nondeterministic specifications are unable to verify the correctness properties for the nondeterministic TL2 algorithm.

5 Conclusion

We presented deterministic specifications for two key correctness properties, strict serializability and opacity, in transactional memories. Our deterministic specifications make the model checking procedure for transactional memories complete and efficient. We formalized the notion of nondeterministic transactional memories to capture realistic contention management. We proved that DSTM and TL2 ensure opacity with arbitrary numbers of threads and variables for all possible contention managers.

Acknowledgment. We are thankful to Laurent Doyen for his kind support in checking language inclusion with his antichain based tool.

References

1. A. Cohen, J. O'Leary, A. Pnueli, M. R. Tuttle, and L. Zuck. Verifying correctness of transactional memories. In *FMCAD*, pages 37–44, 2007.
2. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208. Springer, 2006.
3. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.
4. Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI*, 2008. to appear.
5. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC*, pages 303–323, 2005.
6. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.
7. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. ACM Press, 1993.
8. J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.
9. C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, pages 631–653, 1979.
10. W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.
11. M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT*, 2006.
12. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.
13. M. De Wulf, L. Doyen, T.A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *CAV*, pages 17–30. Springer, 2006.